Project Phase 4

Due Date : 3 December 2025

Code of Conduct

All assignments are graded, meaning we expect you to adhere to the academic integrity standards of NYU Abu Dhabi. To avoid any confusion regarding this, we will briefly state what is and isn't allowed when working on an assignment. Any documents and program code that you submit must be fully written by yourself. You can discuss your work with fellow students, as long as these discussions are restricted to general solution techniques. Put differently, these discussions should not be about concrete code you are writing, nor about specific results you wish to submit. When discussing an assignment with others, this should never lead to you possessing the complete or partial solution of others, regardless of whether the solution is in paper or digital form, and independent of who made the solution, meaning you are also not allowed to possess solutions by someone from a different year or course, by someone from another university, or code from the Internet, etc. This also implies that there is never a valid reason to share your code with fellow students, and that there is no valid reason to publish your code online in any form. Every student is responsible for the work they submit. If there is any doubt during the grading about whether a student created the assignment themselves (e.g., if the solution matches that of others), we reserve the option to let the student explain why this is the case. In case doubts remain, or we decide to directly escalate the issue, the suspected violations will be reported to the academic administration according to the policies of NYU Abu Dhabi (see https://students.nyuad.nyu.edu/campus-life/community- standards/policies/academic-integrity/).

Objective

In this phase, you have to upgrade Phase 3 with scheduling capabilities in the server, which should simulate the role of scheduler to provide concurrency for serving several clients. To avoid dealing with memory and process management, assume that you have only one CPU and only one main thread can run on that CPU; then all the requests (threads of phase3) should schedule among them to run on that main thread. All received requests (from one or more clients) will be added to the waiting queue, including all the relevant information that you find necessary for your scheduling algorithm. Each request/task will be removed from the queue by the selection algorithm, executes on the main thread for some time and then exits if done or returns back to the waiting queue for further execution if not done yet. In this context, you have to handle the following requirements:

- If the received request is a shell command, then it can simply be executed by creating a process and call the corresponding system command to execute. Hence, it will complete execution and exit from the first round.

- If the received request is a program, then it will run until it finishes or until the end of the allocated time based on the adopted scheduling algorithm. If the needed execution is done, then it will exit, otherwise it will return back to the queue for further execution rounds from the point it stopped (Not restarting from the beginning).
- Your combined scheduling algorithm must include RR with different quantum based on the rounds and SJRF, similar to the ones explained in class. (you can add priority too)
- For simulation purposes, the program to be executed can simply have a loop with sleep function in it, which will loop “n” times and print the index of each iteration. The value of n represents the amount of work to be done. Each iteration will be considered one value of time (e.g. 2 iterations will be considered 2 seconds or 2 Milliseconds).
- Execution Scenarios: In case the server is handling a request that solely includes a command, then the server should schedule it for one round of execution. In case the server is handling a request with the value n, this means it will simulate the execution time of a process based on the "n" value. If another command or request arrived, the scheduler module should stop the execution of the current task if the scheduling algorithm selects this option based on your set conditions, save its id and the remaining value of "n", and execute the new task. Once it is done, the server should loop over the list of waiting tasks and schedule the one that fits the best based on your scheduling algorithm and remaining amount of execution time. That being said, your server should always keep a data structure that holds all the tasks and their required information.

Detailed Guidelines

Here, you upgrade phase 3 to provide scheduling capabilities in the server which stimulates the role of a scheduler and provides concurrency for serving multiple clients.
The server listens to requests from multiple clients and puts them in the waiting queue. Use the scheduling algorithm to select and run processes from the waiting queue, and add them back to the queue after a certain time or remove the process once done.

The server would also mainly listen to 2 types of commands from the clients:

- If the command is a shell command, the server runs it right away.
- If the command is a program, then it runs it depending on when the scheduling algorithm will select it. (The process of the program command is then added to waiting queue, and the current process in the queue which is selected by the scheduler algorithm would run).

Potentially, you can have three types of threads:

- Main thread (for accepting connections from new clients)
- Scheduler thread (for scheduling threads)
- Client threads

To handle dealing with multiple threads, you can have multiple semaphores in addition to the waiting queue, and you can include all the relevant information (data structures) that you find necessary for your scheduling algorithm.

The aim is to finish the execution of all these processes in the waiting queue efficiently. Once a program finishes execution, it is removed from the queue. Similarly, once a client closes the connection, the server removes all the processes it has sent.

Scheduling Algorithm

During development, you can start with RR or any simple algorithm to make sure the scheduling is working first, and then implement the combined one. The following are more instructions for the implementation of the required combined algorithm:

- You could use a selectively preemptive Round Robin algorithm with different quantum that uses Short-Remaining-Job-First to prioritize processes.
- You could use the quantum of 3 seconds for the first round and 7 seconds for all the remaining rounds for instance.
- Round-Robin algorithm allows for shorter response times, as it passes through all algorithms in a round, giving an allocated time to run for each process and then moving to the rest.
- Having a smaller quantum for the first round allows for all short processes to run and get response quickly, while also allows that the more time-consuming processes also get to run in the round and that they don’t just remain in the queue.
- When combined with the Short-Remaining-Job-First algorithm, it allows the program to select the process with the shortest remaining time to completion to begin first.
- You can consider a round to be done once a process is preempted even if the quantum is not done yet.
- If the command is a shell command, you could assign it a burst time of -1, so it can always have the priority and is run as soon as it is received rather than putting in the queue. Moreover, if the remaining burst times are equal, you could use First-Come-First-Served algorithm, since the processes are added in a queue.

- Same processes can’t be selected two consecutive times if it has the shortest time, unless it is the only process left.
- The scheduler algorithm is selectively preemptive. This is because shell commands are never preempted and they are run in the first round as they are received. But for program commands, they can be preempted if a process with lower burst time arrives, which would stop the current running process and execute this process with shorter burst time.

Your program will respond to and run the following types of requests from the client:

- It can run shell command executions from the client
- It can also run a demo process program from the client, which executes a demo file and mentions in the command argument the value for the seconds it takes for the program to execute (n).
- It can also take any program execution command as input.

For these commands with unknown predicted burst times, the scheduling algorithm itself could assign a default value.

Testing

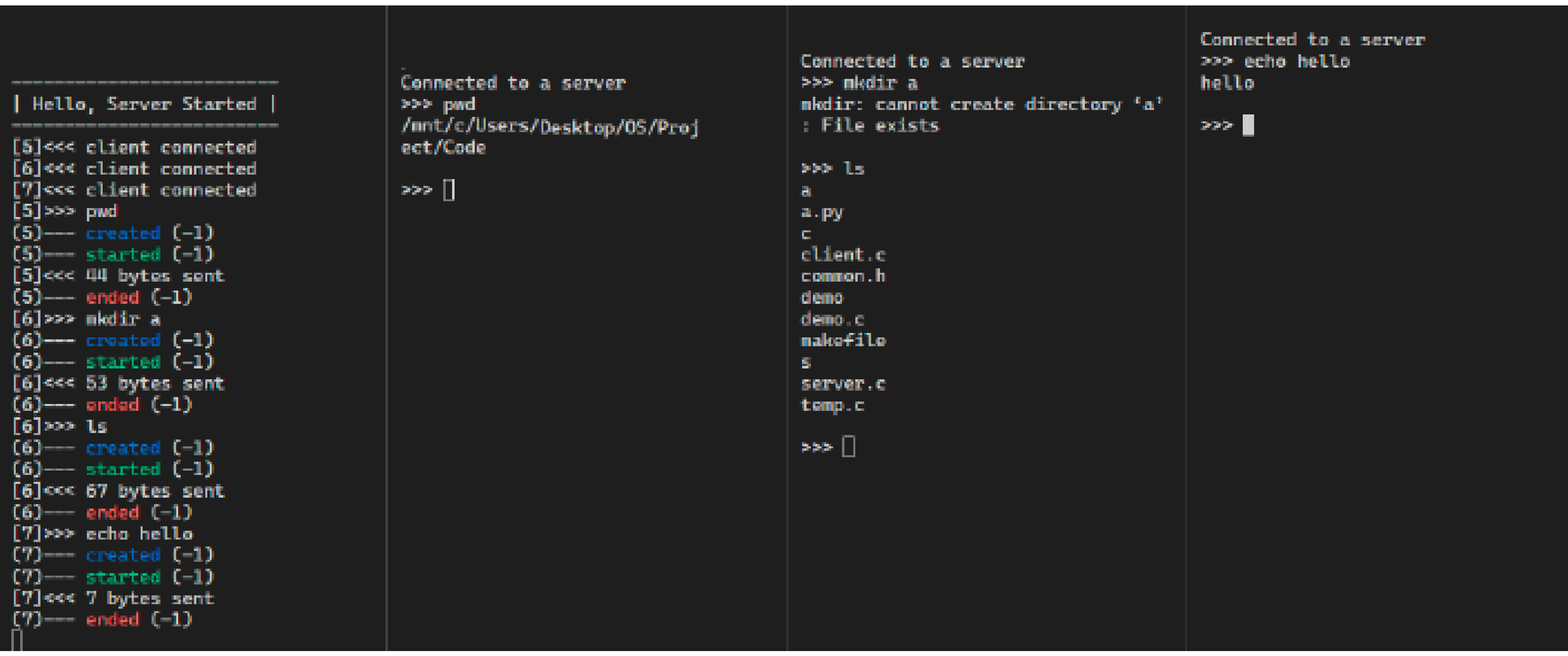
Create a demo program called "demo.c" for testing. The program should accept arguments in the form "demo N", where N is the number of seconds to run. The program must print one output line per second, for a total of N lines. This allows the process to run for the corresponding predicted burst time, enabling the scheduler to schedule the program accordingly.
Note: Output must be printed iteratively (one line per second). Printing the entire output chunk at once is not acceptable, as it does not visually demonstrate the scheduling behavior correctly.

Note: For more details on testing and grading, please refer to this guide at: https://os-project-extra-guidelines.short.gy/

Here are sample screenshots of what it should look like when you run your program:

Test Scenario 1

3 concurrent clients all running single commands (since they are single commands, they should be executed without preemption)



```
-------------------------
| Hello, Server Started |
-------------------------
[5]<<< client connected
[6]<<< client connected
[7]<<< client connected
[5]>>> pwd
(5)--- created (-1)
(5)--- started (-1)
[5]<<< 44 bytes sent
(5)--- ended (-1)
[6]>>> mkdir a
(6)--- created (-1)
(6)--- started (-1)
[6]<<< 53 bytes sent
(6)--- ended (-1)
[6]>>> ls
(6)--- created (-1)
(6)--- started (-1)
[6]<<< 67 bytes sent
(6)--- ended (-1)
[7]>>> echo hello
(7)--- created (-1)
(7)--- started (-1)
[7]<<< 7 bytes sent
(7)--- ended (-1)
```

```
Connected to a server
>>> pwd
/mnt/c/Users/Desktop/OS/Proj
ect/Code

>>> 
```

```
Connected to a server
>>> mkdir a
mkdir: cannot create directory 'a'
: File exists

>>> ls
a
a.py
c
client.c
common.h
demo
demo.c
makefile
s
server.c
temp.c

>>> 
```

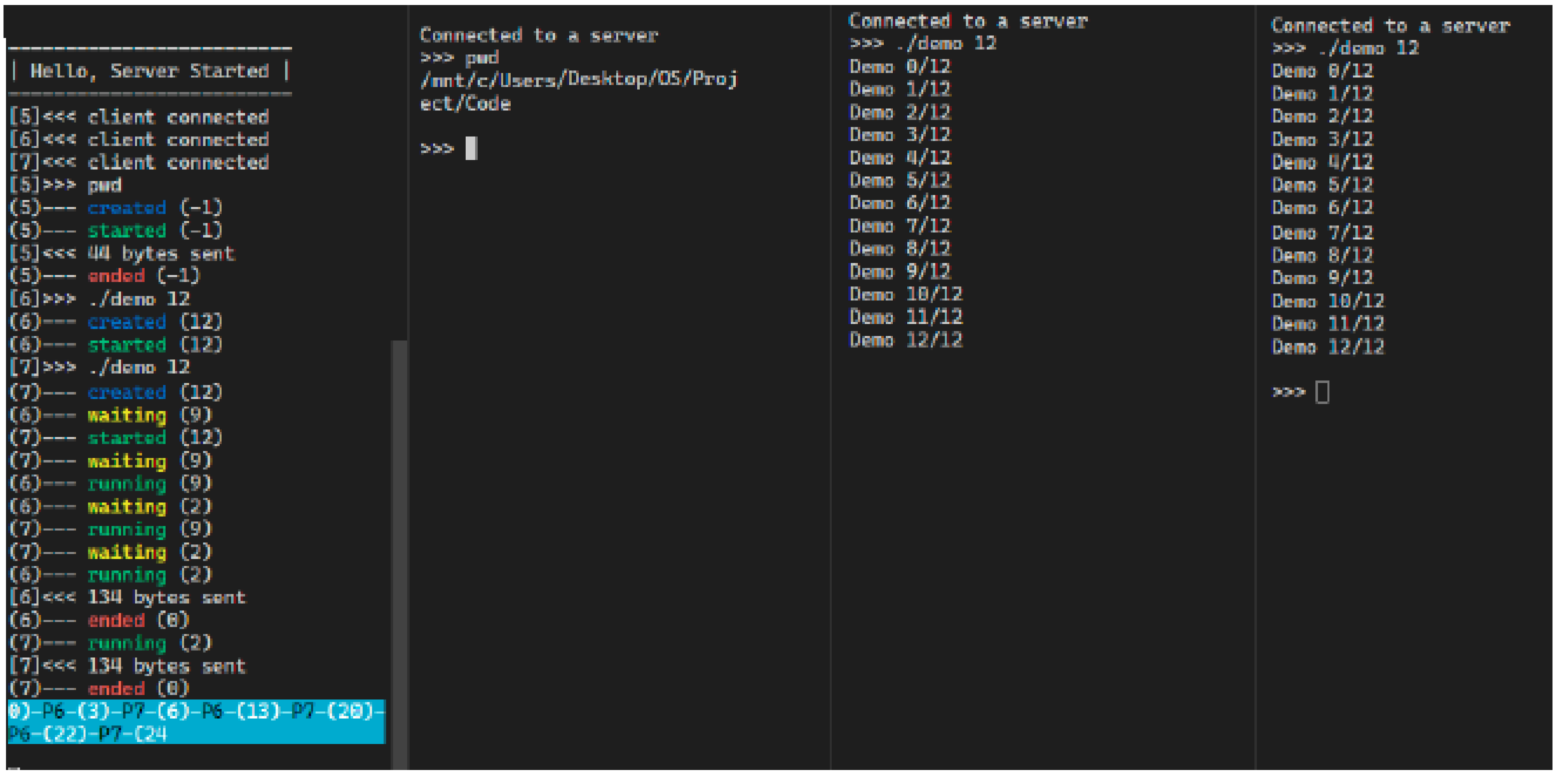
```
Connected to a server
>>> echo hello
hello

>>> 
```

Notice the output of the server that shows a clear detailed log of the concurrent execution of the connected clients, including the number of the client (5,6,or 7 in the screenshot), when each one is created, started, ended, etc...

Test Scenario 2

3 concurrent clients: one client running a single command and the two others running the demo program



```
-------------------------
| Hello, Server Started |
-------------------------
[5]<<< client connected
[6]<<< client connected
[7]<<< client connected
[5]>>> pwd
(5)--- created (-1)
(5)--- started (-1)
[5]<<< 44 bytes sent
(5)--- ended (-1)
[6]>>> ./demo 12
(6)--- created (12)
(6)--- started (12)
[7]>>> ./demo 12
(7)--- created (12)
(6)--- waiting (9)
(7)--- started (12)
(7)--- waiting (9)
(6)--- running (9)
(6)--- waiting (2)
(7)--- running (9)
(7)--- waiting (2)
(6)--- running (2)
[6]<<< 134 bytes sent
(6)--- ended (0)
(7)--- running (2)
[7]<<< 134 bytes sent
(7)--- ended (0)
0)-P6-(3)-P7-(6)-P6-(13)-P7-(20)-
P6-(22)-P7-(24
```

```
Connected to a server
>>> pwd
/mnt/c/Users/Desktop/OS/Proj
ect/Code

>>>
```

```
Connected to a server
>>> ./demo 12
Demo 0/12
Demo 1/12
Demo 2/12
Demo 3/12
Demo 4/12
Demo 5/12
Demo 6/12
Demo 7/12
Demo 8/12
Demo 9/12
Demo 10/12
Demo 11/12
Demo 12/12
```

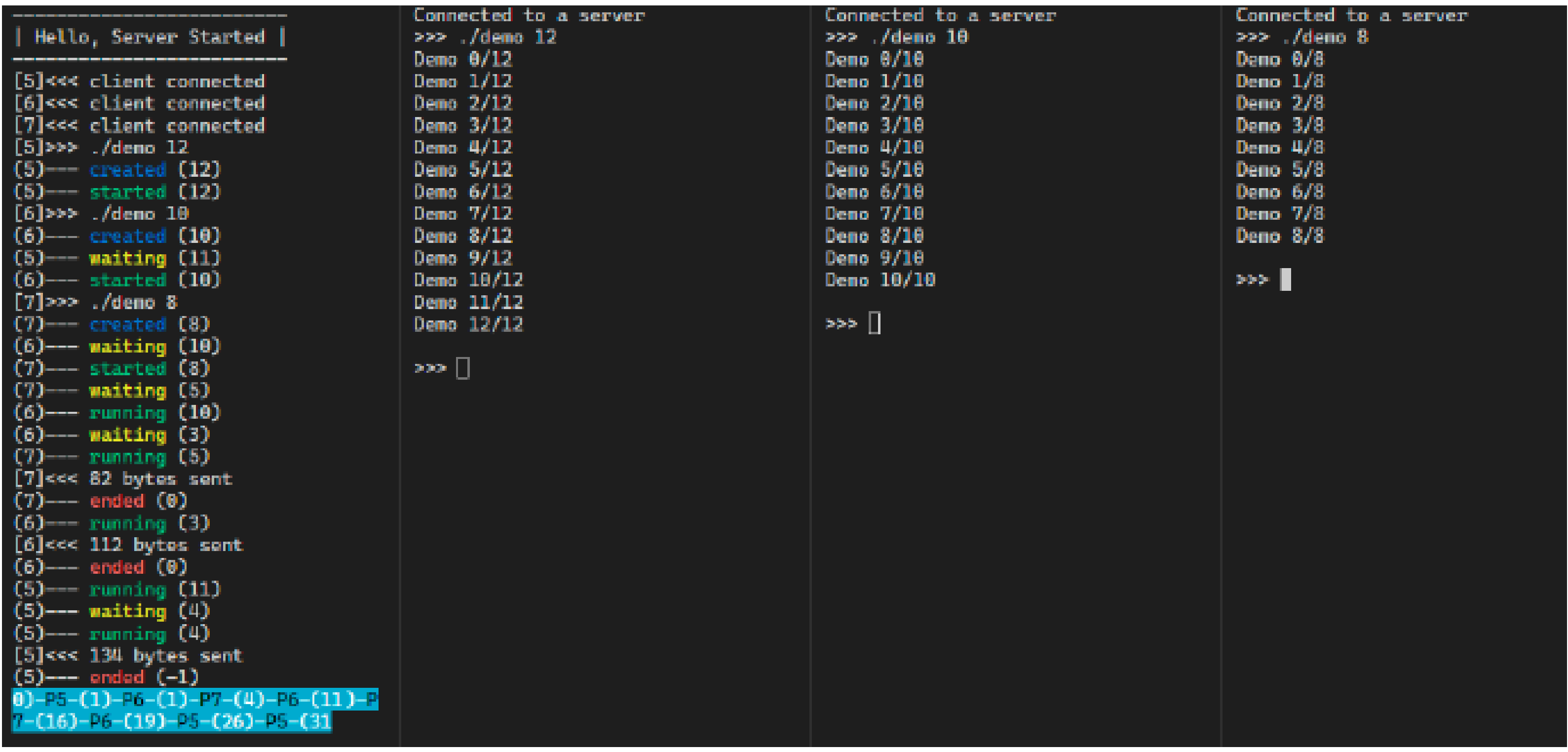
```
Connected to a server
>>> ./demo 12
Demo 0/12
Demo 1/12
Demo 2/12
Demo 3/12
Demo 4/12
Demo 5/12
Demo 6/12
Demo 7/12
Demo 8/12
Demo 9/12
Demo 10/12
Demo 11/12
Demo 12/12

>>>
```

Again, notice the output of the server that demonstrates the scheduling algorithm in action. The log is showing the numbers of connected clients (5,6,7 in the screenshot), when each one is created, started, waiting, running, ended, the time/remaining time for each, etc.... Also, notice the summary shown at the end (the blue highlighted text at the end).

Test Scenario 3

3 concurrent clients: each client running the demo program with different values of N



```
-------------------------
| Hello, Server Started |
-------------------------
[5]<<< client connected
[6]<<< client connected
[7]<<< client connected
[5]>>> ./demo 12
(5)---- created (12)
(5)---- started (12)
[6]>>> ./demo 10
(6)---- created (10)
(5)---- waiting (11)
(6)---- started (10)
[7]>>> ./demo 8
(7)---- created (8)
(6)---- waiting (10)
(7)---- started (8)
(7)---- waiting (5)
(6)---- running (10)
(6)---- waiting (3)
(7)---- running (5)
[7]<<< 82 bytes sent
(7)---- ended (0)
(6)---- running (3)
[6]<<< 112 bytes sent
(6)---- ended (0)
(5)---- running (11)
(5)---- waiting (4)
(5)---- running (4)
[5]<<< 134 bytes sent
(5)---- ended (-1)
0)-P5-(1)-P6-(1)-P7-(4)-P6-(11)-P
7-(16)-P6-(19)-P5-(26)-P5-(31
```

```
Connected to a server
>>> ./demo 12
Demo 0/12
Demo 1/12
Demo 2/12
Demo 3/12
Demo 4/12
Demo 5/12
Demo 6/12
Demo 7/12
Demo 8/12
Demo 9/12
Demo 10/12
Demo 11/12
Demo 12/12

>>> 
```

```
Connected to a server
>>> ./demo 10
Demo 0/10
Demo 1/10
Demo 2/10
Demo 3/10
Demo 4/10
Demo 5/10
Demo 6/10
Demo 7/10
Demo 8/10
Demo 9/10
Demo 10/10

>>> 
```

```
Connected to a server
>>> ./demo 8
Demo 0/8
Demo 1/8
Demo 2/8
Demo 3/8
Demo 4/8
Demo 5/8
Demo 6/8
Demo 7/8
Demo 8/8

>>> 
```

Report Guidelines

Section	Description
Title Page	Includes the phase number, group member names, and NetIDs.
Architecture and Design	Describes the high-level structure of the system, key design decisions, reasons for using specific algorithms/data structures, file structure, and code organization.
Implementation Highlights	Explains the core functionalities, including important functions, algorithms, and logic. Includes references to critical code snippets. Also describes how errors and edge cases were handled.
Execution Instructions	Provides instructions on how to compile and run the program.
Testing	Lists the test cases performed. Includes screenshots and text explanations detailing the tests conducted and the output received.
Challenges	Describes any difficulties and challenges encountered during development and explains how they were resolved.
Division of Tasks	Clarify how the tasks/responsibilities were divided across both team members
References	Cites any resources used during development.

Grading Rubric

Description	Points (/25)
Successful compilation with a Makefile on remote Linux Server	1
Scheduling functionality following the requirements (with output format following required format given in screenshots)	15
Error Handling	1.5
Detailed Comments	1.5
Code modularity, quality, efficiency and organization	1.5
Report	1.5
Demo Video Recording (showing all test cases required)	3

Noteworthy Points

- Each phase is cumulative: all functionality from previous phases must persist and remain correct.

- Feedback comments from previous phases must be addressed; unresolved issues will carry forward as bugs and lead to repeated deductions.

- Although it is understood that you may exchange ideas on how to make things work and seek advice from your fellow students, sharing of code is not allowed.
- If you use code that is not your own, you will have to provide appropriate citation (i.e., explicitly state where you found the code). Otherwise, plagiarism questions may ensue. Regardless, you have to fully understand what and how such pieces of code do.

- Make sure to use separate compilation and apply best practices in your code submission. Aim for high efficiency, reduction of redundant code, and scalability to facilitate updates in future phases.

- Extensive and proper commenting must be done for all the program. Avoid short comments before huge code blocks. They should be meaningful, detailed and explaining all your code logic.

- Your submission **must** be working on the **remote Linux server** successfully.

- For the **demo video recording**: Both team members must be displayed on the recording screen as a camera overlay over the open terminal windows (please make sure the terminal screen is clear and that the camera overlap is at a bottom corner for clarity). Clear audio of both team members should be present as well explaining what you are doing , the test cases you are demonstrating, brief explanation of the output you are getting, etc... This testing must be done on the Linux remote server (There are many tools to do this such as Zoom, Loom Screen Recorder, MmHmm video recording app, etc..) You can check a sample of what this looks like here.

- Submissions:

  A .zip file containing the following:

  - C files + Make file
  - Report (Must be in PDF format)